What Makes An Optimization Problem Hard?

SFI-TR-95-05-046

William G. Macready (wgm@santafe.edu)
David H. Wolpert (dhw@santafe.edu)
The Santa Fe Institute
1399 Hyde Park Road
Santa Fe, NM, 87501

February 8, 1996

Abstract

We address the question: "Are some classes of combinatorial optimization problems intrinsically harder than others, without regard to the algorithm one uses, or can difficulty only be assessed relative to particular algorithms?" We provide a measure of the hardness of a particular optimization problem for a particular optimization algorithm and present two algorithm-independent quantities that use this measure to provide answers to our question. In the first of these we average hardness over all possible algorithms and show that according to this quantity, there are no distinctions between optimization problems. In this sense no problems are intrinsically harder than others. For the second quantity, rather than average over all algorithms, we consider the level of hardness of a problem (or class of problems) for the optimal algorithm. By this criteria there are classes of problems that are intrinsically harder than others.

1 Introduction

The optimization task consists in locating global extrema of a mapping, or cost function. This task has become an important aspect of the new science of "complexity". For example, a common problem in complexity is finding simple local rules of interactions between components that result in some desired emergent collective phenomena. An important part of solving such problems involves defining a measure of how far any particular emergent dynamics is from the desired dynamics and then minimizing that difference. The problem becomes one of optimization. In addition to its importance in the field of complexity, optimization is also a crucial concern in many fields of science and engineering.

To solve optimization problems effective algorithms must be constructed, algorithms which find global or near-global extrema reliably and quickly. Accordingly, it is very important to identify how difficult it is to construct such optimization algorithms for particular optimization problems, *i.e.*, for particular cost functions. This paper is an analysis of this issue from a different point of view than traditional computational complexity.

We say that a cost function is "hard" for a particular algorithm if that algorithm can not quickly locate a near-extremum of that cost function. A natural question then is whether some classes of cost functions are intrinsically harder than others in some algorithm-independent sense. The alternative is that the hardness of a class of cost functions can only be measured relative to a particular optimization algorithm.

As a first step at answering this question, we formally define a hardness measure. This quantity tells us how well any particular algorithm has performed on any particular cost function after some fixed number of iterations of the algorithm. Smaller hardness measures indicate better performance of the algorithm. Given the myriad ways one might wish to use an optimization algorithm, determining a good measure of hardness is not at all trivial. We believe ours to be quite reasonable, and in particular it avoids some problems we have identified in alternative measures. Nonetheless, it may be that there are other reasonable measures besides ours which give different results from those associated with our measure.

Given our measure, we examine two ways to characterize the hardness of a class of cost functions in an algorithm-independent way. In the first we average our hardness measure over all possible algorithms on the class of functions of interest. As we show below, such an expected hardness is identical for any two classes of cost functions. So by this measure, no set of optimization problems is harder than another, and to distinguish between the two sets of optimization tasks we must resort to a comparison of how well particular algorithms perform over the two sets.

A second method of characterizing hardness of a class of cost functions bears similarity to traditional computational complexity theory. In this second approach we first find the algorithm that minimizes the average hardness over our class of cost functions, and in this sense is "optimal" for that class. We then use that minimal average hardness as our characterization of the hardness of the class. As we show below, according to this second approach, there are classes of cost functions that are intrinsically harder than others.

Readers familiar with theory of complexity will note a similarity between the topics addressed in this paper and many of the the issues of computational complexity. Our approach to these topics however is quite different and arises naturally in the context of a preliminary mathematical theory of search we have developed [1].

We begin in Section 2 by considering appropriate measures of gauging the performance of an algorithm on a particular cost function or across a set of cost functions. We point out the difficulty in defining such a measure and motivate the measure we suggest in this paper. In Section 3 we go on to use our measure of performance to characterize the difficulty of a class of cost functions independent of any algorithm. We do this by averaging the performance measure across the class of problems and across all algorithms. We prove that this average is the same for all classes of cost functions so that by this measure no class of problems can be called more difficult than another. In Section 4 we refine our analysis by comparing, for different classes of cost functions, the hardness of that class for the single optimal algorithm for that class. According to this measure there *are* some classes of problems that are intrinsically harder than others. We end by comparing our work to traditional computational complexity in Section 5. Conclusions and open issues are discussed in Section 6.

2 What is hard?

Our notation follows that used in [1]. Formally, the optimization problem consists in locating the global extrema of a single valued mapping $f : \mathcal{X} \to \mathcal{Y}$, which we call the "cost function". We assume $\mathcal{X}$ and $\mathcal{Y}$ are finite and of size $|\mathcal{X}|$ and $|\mathcal{Y}|$ respectively. Without loss of generality we assume we are seeking the global minima of f.

We call an ordered sample of m distinct points from the cost function a "population" of size m and denote it by d_m. In this paper we consider "search algorithms" like simulated annealing [2], genetic algorithms [3], hill-climbing, *etc.* Any algorithm that re-visits points in $\mathcal{X}$ can be compressed into a more efficient one by eliminating the wasted effort of resampling points from f that have already been seen. This point is considered in greater detail in [1]. Thus we consider algorithms that generate a new distinct point at each iteration. In this case the output of m iterations of the algorithm is a population d_m. Populations are ordered according to the time at which the algorithm generated the points.

For convenience we write $d_m \equiv \{d_m(i)\} \equiv \{d_m^x(i), d_m^y(i)\}$ where d_m^x and d_m^y are respectively the $\mathcal{X}$ and $\mathcal{Y}$ components of the population. $\mathcal{D}_m$ is the set of all populations of size m and $\mathcal{D} = \cup_m \mathcal{D}_m$ is the set of populations of all sizes.

Search algorithms of the type considered in this paper rely on extrapolating from n distinct samples of the cost function, d_n, to a new point $x \in \mathcal{X}$. So for the purposes of this paper a search algorithms is a large list which for each possible population indicates the new point in $\mathcal{X}$ to sample. To simplify our exposition we impose the restriction that the new point is selected deterministically (for the same population the algorithm always chooses the same "new point"). However, as discussed in [1] most of our results apply equally well to algorithms which are stochastic and even to algorithms that re-visit the elements of the population. (We frame our analysis in such a way that the extension to stochastic algorithms will be clear.) Given these restrictions, an algorithm is a mapping $a : d \in \mathcal{D} \to \{x \mid x \notin d^x\}$.

This definition encompasses a wide class of search methods. As an example, genetic algorithms may appear not to fall into this class because they select more than one new $\mathcal{X}$ value at a time. However in fact they are in this class because they can be decomposed into a sequence of shorter steps each of which picks only a single new x.

We wish to compare algorithms with the same number of distinct cost evaluations, m (*i.e.*, with the same size populations). This type of comparison is the one most often used in the optimization literature. Usually we are not interested in the time-dependent nature of the algorithm's output, but rather simply in the cost values obtained. Accordingly, in this paper we consider the histogram $\vec{c}$ of the numbers of times each cost value occurs in the population generated by running the algorithm on the cost function for m steps[1]. When time-dependencies (*e.g.* how quickly during the m samples did the algorithm approach the optima) are not of interest, the value of $\vec{c}$ fixes the value of any measure one might use to judge the effectiveness of the search. In particular, it fixes the minimum cost value present in the population. When time-dependencies are important we can consider not the histogram $\vec{c}$, but rather the time-ordered population, d_m. As can be seen from the structure of our proofs our results are equally applicable to these time-dependent notions of hardness.

We seek a performance or "hardness" function, ϕ, of $\vec{c}$ and f, which measures the quality of the search that produced $\vec{c}$ on cost function f. Note that since we are restricting ourselves to deterministic algorithms, the value of ϕ is equally as well specified by (a, f, m) as by $(\vec{c}, f)$.

Unfortunately, there are scenarios which do not seem to have a preferred natural hardness measure. For example, say there are four possible cost values, which we list as $\{y_1, y_2, y_3, y_4\}$ where $y_1 < y_2 < y_3 < y_4$. Say that for a cost function f_1 the fractions of $x \in \mathcal{X}$ such that $f(x) = \{y_1, y_2, y_3, y_4\}$ are 0.1, 0.1, 0.4, and 0.4, respectively. However for another f_2, the four fractions are 0.4, 0.4, 0.1, and 0.1. Imagine that minimization algorithm 1 run on f_1 gets down to y_3, while algorithm 2 run on f_2 gets down to y_2. Which algorithm performed better? Algorithm 2 got to a lower cost value than algorithm 1, but for algorithm 1, there are only 20% of the $x \in \mathcal{X}$ with a lower cost, whereas for algorithm 2 there are 40%. There is no obvious way to decide which algorithm did better. As another, more general example of how

[1] $\vec{c}$ can be represented as a vector of length $|\mathcal{Y}|$ whose y'th component is given by $\vec{c}_{y\in\mathcal{Y}} \equiv \sum_{i=1}^{m} \delta(d_m^y(i), y)$, where $\delta(.,.)$ is the Kronecker delta function.

difficult it is to define hardness, how should one compare search across a largely flat cost function with search across one which has no plateaus?

The traditional computational complexity approach avoids these issues by solely considering how close the algorithm got to the global minimum, in this case y_1. However this measure varies with monotonic transformations of $\mathcal{Y}$. It is also subject to the difficulties listed above. So in many cases it is clearly an inappropriate measure of hardness.

In this paper, to circumvent the difficulties listed above, when comparing hardness we restrict attention to sets of f's all of which are in the same equivalence class. To define our equivalence structure let $\vec{\omega}(f)$ be a vector of length $|\mathcal{Y}|$ whose i'th component is the number of points $x \in \mathcal{X}$ such that $f(x)$ is the i'th y value, y_i. $\vec{\omega}(f)$ is the histogram formed from the $\mathcal{Y}$ values of f. Define $\vec{\Omega}(f)$ as the vector — whose dimension can vary with f — of the successive nonzero components of $\vec{\omega}(f)$. (*E.g.*, if $\omega = \{0,3,2,0,4\}$ the corresponding $\Omega = \{3,2,4\}$.) Then f_1 and f_2 are in the same class if and only if $\vec{\Omega}(f_1) = \vec{\Omega}(f_2)$. As an example, if f_1 is f_2 "shifted" upwards, they are in the same class. Similarly f_1 is in the same class as f_2 if it is produced from f_2 by any monotonic transformation of the cost values.

Even given this restriction on the set of f's we will consider, it is not immediately obvious what the measure of performance, ϕ, should be. One possible measure (of interest in computational complexity) is the minimal y such that $\vec{c}_y$ is nonzero, which we write as $\phi(\vec{c}, f) = \min(\vec{c})$. We feel that this measure is a poor indicator of the quality of the search however, because it is not invariant under a translation of the $\mathcal{Y}$ values. (Note that such translations do not change the equivalence class.) To address this shortcoming we might instead define ϕ in terms of $|\min(\vec{c}) - E(f)| \,/\, \sigma(f)$, where $E(f)$ is the average (across $\mathcal{X}$) cost value of f, and $\sigma(f)$ is the standard deviation of these cost values. With this ϕ we would be measuring the number of standard deviations between $\min(\vec{c})$ and $E(f)$, the average cost in f. However this measure is also not without difficulties. For example, algorithms would be judged to perform better on f's for which the algorithm produces the same $\vec{c}$ and for which $E(f)$ is the same, but which have smaller standard deviations. It is not clear that we wish to favor such f.

To avoid these problems we define ϕ to be the "statistical weight" of the set of $x \in \mathcal{X}$ such that $f(x) < \min(\vec{c})$. Stated another way, our measure is the fraction of points in $\mathcal{X}$ which have lower cost values than the minimum cost value found by the algorithm. Formally

$$\phi(\vec{c}, f) = \phi(a, f, m) \equiv 1 - \frac{\sum_{x\in\mathcal{X}: f(x) > \min(\vec{c})}(1)}{|\mathcal{X}| - \mu(f)}, \qquad (1)$$

where $\mu(f)$ is the number of distinct $x \in \mathcal{X}$ that give the same global minimum of f (the degeneracy). By $\sum_{x\in\mathcal{X}: f(x) > \min(\vec{c})}$ we mean a sum over all x values such that $f(x) > \min(\vec{c})$. This ϕ is normalized so that it equals 1 if $\min(\vec{c})$ is the global maximum and equals 0 if $\min(\vec{c})$ is the global minimum.

We next turn to the question of measuring the hardness, Φ, of a class of cost functions, or more generally of a distribution over cost functions, for a particular algorithm, a. There are a number of ways to do this. The simplest choice is the average performance of a according to the distribution at hand,

$$\Phi_1(a, m) \equiv \sum_f \phi(a, f, m) P(f) \,. \tag{2}$$

where $P(f)$ is the probability that a particular cost function f is in our distribution. (Since our comparisons are only valid within an equivalence class, in Equation 2 the support[2] of $P(f)$ is implicitly assumed to lie entirely within an equivalence class.)

There are a number of alternative choices. For example, we might measure the difficulty of a class of cost functions, F, as the difficulty of the hardest instance within the class:

$$\Phi_2(a, m) \equiv \max_{f \in F} \phi(a, f, m) \,. \tag{3}$$

This conservative measure might be appropriate when one is unsure of $P(f)$ except that its support is F. We will focus on measure Φ_1.

Having found a reasonable scheme for measuring the hardness of an f or a $P(f)$ for a particular algorithm, we consider the intrinsic hardness of $P(f)$ itself. The intrinsic hardness of a class of problems $P(f)$ measures hardness without any *a priori* specification of the algorithm. Determination of intrinsic hardness is taken up in the next two sections.

3 Average behavior over all algorithms

Perhaps the simplest algorithm-independent measure of hardness of a class of problems, $P(f)$, is the uniform average of $\Phi_1(a, m)$ over all possible algorithms, a.

$$\sum_a \Phi_1(a, m) = \sum_f P(f) \sum_a \phi(a, f, m) \tag{4}$$

It turns out that this measure is independent of $P(f)$. To prove this we prove that the innermost sum over algorithms is independent of f, for all f in the same equivalence class. (Recall our implicit assumption that $P(f)$ has its support in a single $\vec{\Omega}$

[2]The support of a probability distribution over a set $\mathcal{S}$ is the subset of $\mathcal{S}$ which has non-zero probability.

equivalence class). To do this, we replace our sum over algorithms with a sum over all possible orderings of points sampled from $\mathcal{X}$. To that end, first note that since $|\mathcal{X}|$ and $|\mathcal{Y}|$ are finite, populations are finite, and therefore any deterministic a is a finite list indexed by all possible d's (including the empty d and excluding those d's that extend over the entire input space). Each entry in the list is the x the algorithm a outputs for that d-index.

Consider any particular ordered set of m distinct x values. Such a set is an "ordered path" π_m. (Note that π_m is the $\mathcal{X}$ component of a population, d_m^x, but for clarity we avoid referring to it as such.) Letting "$\{a : a_m(f) = \pi_m\}$" specify the set of algorithms which produce populations d_m when run on f such that $d_m^x = \pi_m$, we write

$$\sum_a \phi(a,f,m) \;=\; \sum_{\pi_m} \sum_{a:a_m(f)=\pi_m} \phi(a,f,m).$$

We rewrite this as

$$\sum_{x_1} \sum_{x_2 \notin \{x_1\}} \cdots \sum_{x_m \notin \{x_1,\cdots,x_{m-1}\}} \; \sum_{a:a_m(f)=\{x_1,\ldots,x_m\}} \phi(\{x_1,\cdots,x_m\},f,m),$$

with the obvious definition that $\phi(\{x_1,\cdots,x_m\},f,m) \equiv \phi(\vec{c},f)$ where $\vec{c}$ is the histogram made from the values $\{f(x_1),\cdots f(x_m)\}$.

The summand is independent of the innermost sum and regardless of $\{x_1,...,x_m\}$ or f, there are the same number of terms in the sum[3]. Inserting the definition of ϕ establishes the following:

$$\begin{aligned}\sum_a \Phi(a,m) \;&\propto\; \sum_{x_1} \sum_{x_2 \notin \{x_1\}} \cdots \sum_{x_m \notin \{x_1,\cdots,x_{m-1}\}} \phi(x_1,\cdots,x_m,f,m) \\ &=\; \sum_{x_1} \sum_{x_2 \notin \{x_1\}} \cdots \sum_{x_m \notin \{x_1,\cdots,x_{m-1}\}} \left(1 - \frac{\sum_{x\in\mathcal{X}:f(x)>\min_i f(x_i)}(1)}{|\mathcal{X}| - \mu(f)}\right)\end{aligned}$$

where the proportionality constant is independent of f.

Since $\mu(f)$ is the same for all f within the same equivalence class, the only possible dependence on f in the above equation arises in the expression:

$$\sum_{x_1} \sum_{x_2 \notin \{x_1\}} \cdots \sum_{x_m \notin \{x_1,\cdots,x_{m-1}\}} \; \sum_{x\in\mathcal{X}:f(x)>\min_i f(x_i)} (1). \tag{5}$$

Given f, define a bijection $x \in \mathcal{X} \leftrightarrow \{z^{(1)},\cdots,z^{(|\mathcal{X}|)}\}$. As shorthand, write "$f(z^{(i)})$" to indicate f of the x value corresponding to $z^{(i)}$. Require of our bijection

[3]This follows immediately from viewing an algorithm as a list (see above); the restriction on a in the sum fixes a total of m entries in that list, and has no effect on the remaining entries.

that for all i, j $f(z^{(j>i)}) > f(z^{(i)})$, where the mapping to z values from any set of x's having the same $f(x)$ is according to a prefixed f-independent tie-breaking scheme. Intuitively, the $z^{(i)}$ are simply the x_i indexed in order of non-decreasing $f(x)$.

Define Z as the set of all $z^{(i)}$. View the sequences over $z^{(1)}, \cdots, z^{(|\mathcal{X}|)}$ as consisting of contiguous blocks of values of z within which $f(z)$ is constant. Let $\xi(z)$ index the first element of the block that z is in. Note that ξ will be the same for all f in the same equivalence class.

Now since the mapping from $\mathcal{X}$ to Z is a bijection, we can replace the sums over x_i in Equation 5 by sums over $z_i \in Z$ and obtain

$$\sum_{z_1} \sum_{z_2 \notin \{z_1\}} \cdots \sum_{z_m \notin \{z_1 \cdots z_{m-1}\}} \sum_{z : \xi(z) > min_i \xi(z_i)} (1) \tag{6}$$

By inspection, this result is independent of f for all f in the same equivalence class.

This establishes that the average over a of $\phi(a, f, m)$ is the same for any f's in the same equivalence class. Thus the average over a of $\Phi_1(a, m)$ is independent of $P(f)$ for all $P(f)$ restricting f's to the same equivalence class. Note that no property of $\phi(\vec{c}, f)$ was used in proving this result. Therefore this result concerning Φ_1 holds for any hardness measure ϕ that is based solely on $\vec{c}$ and f. (Note though that whereas Φ_1 makes no distinction between different $P(f)$, Φ_2 may if $\sum_a$ and $\max_{f \in F}$ do not commute.)

Subject to the caveats discussed above, any problem, class of problems, or distribution of problems is as hard as any other, when hardness is averaged over all possible algorithms. To some, such lack of distinguishability may seem intuitively obvious. However, indistinguishability does not hold when we compare the performance of algorithms that are optimal for their associated $P(f)$ rather than average over algorithms. This is true even with the same equivalence class restrictions on $P(f)$'s that were made above, as the analysis in the next section demonstrates.

4 Optimal behavior

In the previous section, we measured the algorithm-independent hardness of a $P(f)$ restricted to an equivalence class by averaging $\Phi_1(a, m)$ over all a. Such an average can be misleading because although $\sum_a \Phi_1(a, m)$ is the same for all such $P(f)$, the probability of any particular value of $\Phi_1(a, m)$ need not be. For example, it may be that for a small set of a, $P_1(f)$ has a much higher $\Phi_1(a, m)$ (and is therefore harder for those a) than does $P_2(f)$. To fulfill the equal average requirement of the previous section, $\Phi_1(a, m)$ must then be slightly larger for $P_2(f)$ than it is for $P_1(f)$ for the large set of the remaining algorithms. In such a scenario, one might argue that $P_1(f)$

is "intrinsically harder" than $P_2(f)$, since in the worst case (over algorithms) it is much harder than $P_2(f)$.

There are many pther ways that the mapping from a to $\Phi_1(a, m)$ can be used to distinguish the hardness of $P(f)$'s. In this section we examine the following one related to the distinction discussed in the paragraph above: Let $\bar{a}_m(P(f))$ be the best possible algorithm for $P(f)$, for populations of size m. Formally, $\bar{a}_m(P(f)) \equiv \text{argmin}_a \Phi_1(a, m)$. (Recall that Φ_1 is a function of $P(f)$ — see Equation (2).) Then we can define $\bar{\Phi}_m(P(f))$ as the hardness of $P(f)$ for $\bar{a}_m(P(f))$: $\bar{\Phi}_m(P(f)) \equiv \Phi_1(\bar{a}_m(P(f)), m) = \min_a \Phi_1(a, m)$. So $\bar{\Phi}_m(P(f))$ is how hard $P(f)$ is for the best algorithm for $P(f)$.

For simplicity, restrict attention to those $P(f)$ that are uniform over some set of cost functions F and zero elsewhere. Let $|F|$ denote the number of cost functions in the support of $P(f)$.

We wish to characterize the equivalence classes of those F sharing the same $\bar{\Phi}_m(F)$ for certain m. (Each such class is a subset of the equivalence classes discussed in Section 2.) Here we won't characterize these classes in full, but rather will analyze them for certain small values of m. Our analysis will involve explicit construction of optimal algorithms. These constructions will depend upon m, $|F|$, and the location of the global minima of each $f \in F$. We proceed by considering each of these dependencies.

$|F| = 1$, any m: In this case our class of functions contains only a single instance, f. The optimal algorithm for this problem locates the global minima with the first sample of f simply by starting there. Regardless of f it is always possible to construct this optimal algorithm. So no single F is inherently harder than any other.

$|F| = 2, m = 1$: Next we consider the case where we have two functions in the class and are allowed only a single guess. Here distinctions between classes appear. A pair of f's sharing an $x \in \mathcal{X}$ at which the respective global extrema occur are easier than any pair of f's for which this is not the case. If there is an x at which the extrema coincide the optimal algorithm guesses the shared value achieving maximal performance on both functions, while in the second case the algorithm can maximize performance only one or the other of the cost functions at a time.

$|F| = 2, m = 2$: In this case the algorithm is allowed as many samples as there are cost functions. The optimal algorithm can always attain the extrema of both cost functions by first visiting the extrema of f_1 and then visiting the extrema of f_2. So no pair of problems is intrinsically harder than any other if we are given two cost evaluations. This generalizes whenever $m \geq |F|$.

$|F| = 3, m = 2$: Here we demonstrate through a simple construction that some F's are harder than others. The demonstration is similar to the $F = 2$, $m = 1$ case; qualitatively different behavior arises for the $|F| = 4$ case (see below). Consider one set of cost functions, F_1 for which the global minima on all three functions occurs at the same point x_1. The optimal algorithm identifies the global minima for all three functions by selecting x_1 on either the first or second iteration. For a second

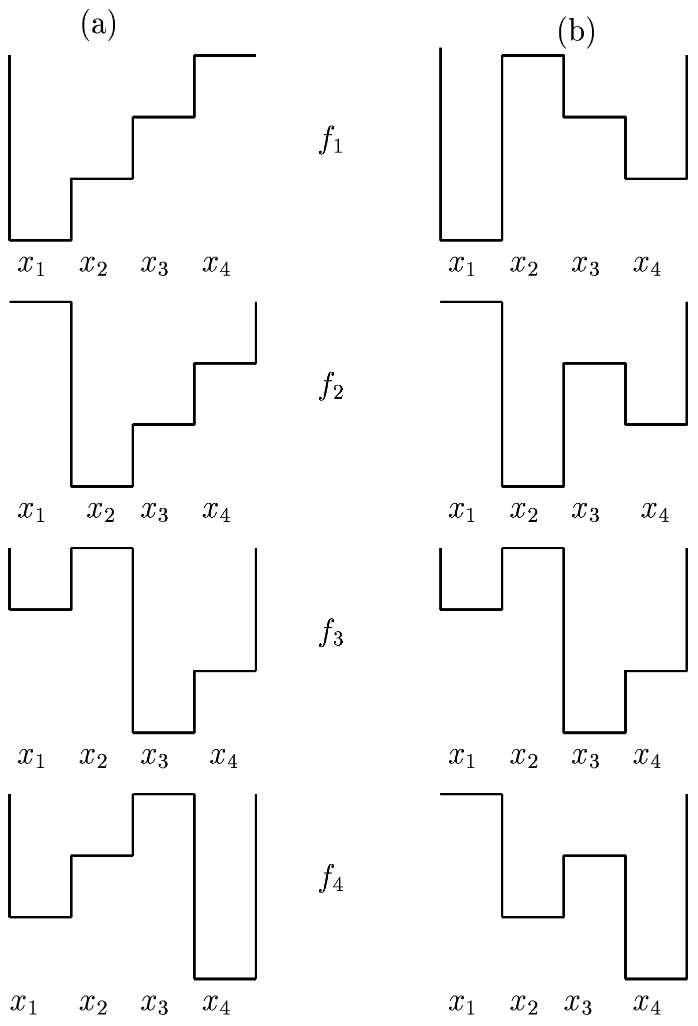

Figure 1: Two sets of four cost functions, f_1, f_2, f_3, f_4 from the same equivalence class with the same optima. The optimal algorithm for each class performs well on (a) but poorly on (b).

set of cost functions F_2, the minima are not coincident, but occur at x_1, x_2, and x_3 respectively. In this case the optimal algorithm can only locate the global minima on two of the three functions.

$|F| = 4, m = 2$: Here we present a more interesting example of two sets of cost functions all lying in the same Ω, where one set is more difficult than the other. In Figure 1(a) we present one set of 4 cost functions f_1, f_2, f_3, f_4, with extrema at x_1, x_2, x_3, and x_4 respectively. For the functions of Figure 1(a) the optimal algorithm identifies the global minima on all 4 functions. If the associated $\mathcal{Y}$ values are $y_1 < y_2 < y_3 < y_4$ the algorithm that accomplishes this is:

```
1: select $x_1$
2: if $f(x_1) = \begin{cases} y_1 & \texttt{select } x_2 \texttt{ or } x_3 \texttt{ or } x_4 \\ y_2 & \texttt{select } x_4 \\ y_3 & \texttt{select } x_3 \\ y_4 & \texttt{select } x_2 \end{cases}$
```

Case (b) of figure 1 appears similar, but here the optimal algorithm can only locate the global minima on 3 of the 4 functions. The difference from Figure 1(a) is that for the functions in Figure 1(a), for any particular x all four function f_1–f_4 have different $\mathcal{Y}$ values at that x, so no matter what x is picked the optimal algorithm knows which function it is on and can identify the minima at the next step. In Figure 1(b) the $\mathcal{Y}$ values at the x points are not all unique so the optimal algorithm can not assuredly identify the function it is on and thus can not assuredly locate the global minimum.

So knowledge of the y value conveys information about the location of the optima in case 1(a) whereas it does not in case 1(b). Intuitively, with the set of functions of Figure 1(b) it is difficult even for the optimal algorithm to do well since the information gained with each guess is not complete. See [4] for a formalization of this notion in the context of supervised learning. To summarize: some classes of cost functions *are* harder than others for optimal algorithms.

Given the close parallel between supervised learning and combinatorial optimization (see [1] and [5]), the results of this section are not too surprising. After all, the (Bayes) optimal supervised learning algorithm performs differently for different priors over target functions; this is the direct analogy of saying that the optimal search algorithm for a $P(f)$ performs with different efficacies for different $P(f)$.

5 Comparison to computational complexity

The question, "how hard is an optimization problem?", has been investigated in theoretical computer science under the guise of computational complexity [6] and information-based complexity [7]. Over the past 20 years a hierarchical classification of problem difficulties has been designed based on a standard computer – the Turing machine. That theory assigns difficulty by determining how the size and running time of a program running on a Turing machine that achieves a particular performance level varies with the size of the problem. Polynomial (P) algorithms scale as some polynomial in the problem size, while non-polynomial (NP) algorithms apparently scale faster than polynomially.

Our approach is quite different, with different motivations. In particular, we do not consider scaling issues – we are only concerned with one problem at a time. Nor do we restrict attention to algorithms with some particular computational power (*e.g.*, Turing machines). Rather than measuring problem difficulty in terms of characteristics of the computational device, our measures of difficulty are "oracle based"; all comparisons are based on the number of calls to the cost function or oracle. (Comparing algorithms by the number of cost function evaluations is a common basis of comparison in the optimization literature.) Moreover we compare based on the number of *distinct* calls to the oracle. Insisting on distinct calls is natural since any algorithm that makes redundant calls can be improved by eliminating these duplicate calls.

In addition, we have designed our framework to be broad enough to consider all cost functions and algorithms simultaneously. It is only by including all cost functions *and* all algorithms within the same framework that one can understand the connection between an optimization problem and an effective algorithm [1]. In contrast, the field of computational complexity analyzes the relationship between particular sets of cost functions and particular associated algorithms. To consider all algorithms and all cost functions in a single framework, we have had to specially design measures of how good a particular population produced by an algorithm is for a particular cost function. Our concern with how to define general measures of how well an algorithm has optimized a cost function has received relatively little attention in the computational complexity.

Finally, our analysis for the case where intrinsic hardness is defined by averaging over algorithms is rarely done in computational complexity. Yet such a measure arises naturally, once one adopts a probabilistic perspective on optimization: In one sense an optimization problem f is known exactly in that for any input, $x \in \mathcal{X}$, we can calculate an output, $y \in \mathcal{Y}$. In spite of this knowledge though, in practice most information about f is effectively unknown (*e.g.* we don't know its extrema). Admitting that we have incomplete knowledge about f, and then trying to make inferences in spite of this, leads naturally to a probabilistic perspective. Though more recent complexity-based research has taken a probabilistic perspective, we are not aware of other work viewing particular problems as probability distributions, $P(f)$.

6 Discussion

This paper is an investigation of the foundations of optimization or the "mathematics of search", from an algorithmic perspective. As such, it forms a natural companion to [1], where we investigated optimization from a "cost function perspective". We have found strong overlap between the algorithmic side of the mathematics of search and computational complexity. However, there are important foundational difference between the two subjects. We have tried to point out how the work described here differs from traditional computational complexity. Readers wishing deeper understanding of our motivations might consult [1].

We close by listing some future research directions. Firstly, there are a number of alternatives for algorithm-independent measures of the hardness of a $P(f)$. Some, briefly mentioned in the text, are quite similar to conventional computational complexity measures. For example, when $P(f)$ is nonzero only over some set F of cost, it often may be appropriate for Φ to be given by the worst performance of a on any of the $f \in F$, *i.e.*, by our $\Phi_2(a, m)$ measure mentioned in Section 2. This is very similar to the worst case analysis of computational complexity. Presumably the results presented here will change using this worst-case definition of hardness. Future work involves exploring such alternative definitions.

It would also be interesting to carry out the analysis of optimal rather than averaged algorithms in more detail. In particular, the notions of information gain, and of the VC dimension and VC entropy [4] of a class $|F|$, may be helpful. Another question we might ask is: Given $|F|$ and m, what fraction of classes are not assuredly solvable for the optimal algorithm? That is, how pervasive are the problems discussed in Section 4? For problems for which $|F| > m$ (otherwise the classes are trivially easy) little is known.

¿From the opposite perspective we can ask questions such as: How common are algorithms that are not optimal for any $P(f)$? (Note that one can use the inverse of this concept — the $P(f)$ that is optimal for a particular algorithm — to measure how close two algorithms are. Two algorithms are close if the corresponding $P(f)$'s are close according to some metric on the space of probability distributions. See [1].) What changes if we restrict ourselves to stochastic algorithms whose first point is uniformly randomly chosen?

Other future work involves perturbing our same-equivalence-class assumption. Perhaps one can define a metric on the space of $\vec{\Omega}$ values, and bound how much hardness varies between two equivalence classes as a function of the distance between their respective $\vec{\Omega}$ values.

Acknowledgments
We would like to thank William Spears for bringing our attention to the problem

addressed by this paper and the referees for useful comments.

References

[1] D.H. Wolpert, W.G. Macready, SFI-TR-95-02-010, submitted to *Oper. Res.*, (1995).

[2] S. Kirkpatrick, C. D. Gelatt Jr., M. P. Vecchi, *Science*, **220**, 671, (1983).

[3] S. Forrest, *Science*, **261**, 872, (1993).

[4] D. Haussler, M. Kearns, R. Schapire, *Machine Learning*, **14**, 83, (1994).

[5] D.H. Wolpert, *The lack of a priori distinctions between learning algorithms*, and *The existence of a priori distinctions between learning algorithms*, submitted to *Neural Computation*, (1995).

[6] C.H. Papadimitriou, *Computational complexity*, Addison-Wesley, (1994).

[7] J.F. Traub, G.W. Wasilkowski, and H. Wozniakowski, *Information-Based Complexity*, Academic Press:New York, (1988).